EN.605.202.81: Introduction to Data Structures

Peter Sullivan

Lab 3 Analysis

Due Date: 8/06/24

Lab 3 Analysis

Data Structures

Before we go through the data structure structures used in this lab, lets quickly summarize what the lab required. The lab assignment requested that we set up a Huffman encoding process that accepts a frequency table, creating a Huffman encoding tree based on the desired priority queue. Then using the Huffman code, we needed to decode and encode messages. There was also the ask to print out the Huffman tree using pre order traversal.

The main data structure used in this analysis was the tree data structure. I created a Huffman class which really acted a tree node class with Huffman encoding methods. The base of the class was initiating a tree node with an additional value called char. I did use a lot of list data structures to hold tree nodes as I processed and sorted them depending on the task required. When decoding and encoding the messages, I returned those messages as strings. I modified and moved those strings as required.

When looking at the tree data structure. You can build a node using the Huffman class. I also set up a build Huffman tree method, that creates Huffman nodes, and recursively creates a tree by slowly processing a list containing nodes. Huffman encoding requires that the list containing the nodes is sorted in a priority queue. To ensure that the priority queue was set up correctly, I utilized lists. Even though I was using lists, I treated them as stacks. I know that we have a lot of different sort methods, but I was pressed for time with the deadline, and stacks are very easy to conceptualize. I used two lists to pop and append depending on the priority that was assigned from the lab.

Data Structure Justification/Implementation

The tree data structure was almost a necessity for this lab. Of course, since this lab was based on Binary Trees, so it was expected that we use a tree data structure. I do believe we could have used a nested list to act as a tree, but that would get very hard to visualize and could become difficult to manage. For example, we could have done [tree 1[tree2[tree3]]]. Tree 3 is in tree 2 and tree 2 is in tree 1. For a tree with many levels, this will become unmanaged very quickly.

Using the tree data structure, I was able to easily traverse complex nested elements. Specifically using a binary tree made the process very easy. There were only two children per tree at most, and to find the leaves, we can recursively move through the nodes starting from the left and working our way to the farthest right leaf.

A main criterion for the Huffman encoding process is building a priority queue that manipulates and moves the nodes depending on the priority criteria. To hold the nodes in place, I found a list data structure as the easiest and most efficient data structure for this task. Utilizing the pop method, I was able to easily grab the front elements of the queue while building the Huffman tree.

In order to sort the priority queue, I had to decide on a data structure and sorting method. There are so many different sorting methods, but I decided to move forward with a simplistic and easy sorting method. If we are using an alphabetic coding sequence, the frequency table will not get larger than 26 items. The files that we are encoding are only roughly 3 to 4 lines each. We have a very little amount of data, and time and space complexity are not an priority. I'm also not as familiar with the tree data structures, so I did not want to choose an additional data structure for sorting that would complicate the program even further.

I decided to use two stacks to create a priority queue. The function found in the Huffman Class takes in a list of nodes with a value, and character. The stacks then slowly push items onto a main stack. As we push items on the main stack, we compare the items currently on the stack with current item pushed from our list. If the items need to be sorted, then we utilize both stacks to push and pop, until we have a sorted stack. The time complexity and space complexity may not be the most efficient, but due to the small amount of data and time constraints to submit this lab, I choose the easiest data structure to work with. The sorted nodes were appended to a stack and the stack was returned before continuing the recursive process of building the Huffman tree.

I believe I did achieve some useful data compression using this method. We utilized the frequency table, which allowed me to identify characters that are used more frequently. Those characters are then assigned with shorter bits per character, and the less frequent ones with longer bits. If we were using conventional encoding as mentioned in the zy book, ASCII characters use 8 bits each. So we would need 8*26 (26 characters in the alphabet) which would give us 208 bits needed. For the Huffman compression algorithm, if I summed up the lengths of the bits assigned to each character in my Huffman code. The total length of all bits came up to 132 bits. So, we did save 76 bits when using the Huffman encoding compression.

Changing the scheme to break ties or giving precedence to alphabetical ordering and then the number of letters in the alphabet would absolutely affect the results. I found that one of my main issues with this lab was that my priority queue wasn't working correctly. If the queue is working incorrectly, the overall structure of the tree is changed. Which means the Huffman code is also changed. Slight changes to the priority queue will impact the lengths of the binary strings that are assigned to each character in the Huffman code. For example, if something is pushed back in the priority queue, they will receive a longer binary string. If we push items that are more frequent in the frequency table to the back on the queue, this will also increase the size of the total number of bits required for the Huffman compression and could diminish the overall efficiency of the compression algorithm.

Complexity

For this analysis, we had two major data structures affecting the efficiency of this algorithm. We had the priority queue that was using two stacks to push and pop the items back in forth based on the priority criteria. We also had a tree data structure that was used when creating the Huffman Tree. We also traversed the Huffman tree and printed out the values/characters assigned with each node in a pre-order traversal. This method of traversal also impacted the speed and efficiency of the program.

We'll first look at my stack data structure, since my build Huffman tree method calls on the stack each time to sort before building the parent node in the current recursive call. This function has a while loop nested in another while loop. Which means our time complexity is O(N^2). The space complexity is fixed to n elements. So, space complexity would be O(N).

Next let's look at our tree data structure used by the build Huffman method. This data structure reads in a list, builds a node, sorts the list, and recursively creates a Huffman tree. For this function, we only read through the list which is O(N), but we do call on the sort function which pushes the time complexity to O(N). The space complexity would still be O(N) for this function.

We have another function called get Huffman codes, this traverses the tree and assigns a prefix to the leaf nodes. The recursive calls on the tree would give us log n time complexity. We do however have to process the list, which would give O(N) complexity as O(N) is larger than O(log N). The space complexity would also be O(N).

The final method that uses the tree data structure is the pre order traversal function. This method traverses the tree and appends the output to a list. The recursive nature of the calls gives the time complexity of O(log N) but we are dealing with a list as the input which will give us a space and time complexity of O(N).

Enhancements

The original lab asked us to print the tree out in pre order traversal. I added in an additional argument that is defaulted to pre, but can be inputted if desired. The Additional argument is the traversal type. If desired, the user can input either Pre, Post, In, or nothing and the output file will change based on the input. If the user inputs Post at the end of their cmd line, they will get a output file that specifies post order traversal, and the tree will be printed out in the post order traversal type. Here are examples of running the script that work for each.

- python -m Lab3 resources\FreqTable.txt resources\Encoded.txt resources\ClearText.txt output\Output.txt
- python -m Lab3 resources\FreqTable.txt resources\Encoded.txt resources\ClearText.txt output\Output2.txt
- python -m Lab3 resources\FreqTable.txt resources\Encoded.txt resources\ClearText.txt output\Output2.txt Post
- python -m Lab3 resources\FreqTable.txt resources\Encoded.txt resources\ClearText.txt output\Output2.txt Pre
- python -m Lab3 resources\FreqTable.txt resources\Encoded.txt resources\ClearText.txt output\Output2.txt In

As mentioned above, there is no need to add the Traversal type. This will always be defaulted to Pre order if ignored.

Lessons Learned

I started this lab way too late. I am going to absolutely start the next lab assignment right away after I submit this assignment. I waited way too long to start this lab. It's a tight schedule for this class, and this lab was much harder for me then the previous lab. I found the tree data structure very difficult to grasp at first. Before this lab, I would mostly represent the tree data structure as a nested list. Each sub list contains info about the node or item. I also did something like linked lists. Obviously, a nested list was not an option for this assignment. If I had started this assignment earlier, I would have been able to add in more fail safes for exceptions.

I found a good way for testing/debugging before putting all the code together and running as a module. I worked in jupyter notebooks and pulled the classes into that notebook. I then mimicked the main and lab3 functions files in the jupyter notebook. Once I had a program similar to running in a modular manner, I then slowly went through the entire process of building a Huffman tree, decoding and encoding messages.

This lab required a large amount of debugging! Nothing seemed to work at first. I had a very difficult time getting the message for 'Hello World' to work. The issue was that I wasn't sorting my priority queue after each time I created a node. Instead, I was sorting once with my priority and then creating my tree. I should have spent a longer time trying to understand Huffman encoding before jumping right into the coding portion. Instead, I had to slowly debug while running through the lecture notes, videos and

the zy book. Next time, I will focus on the question asked. Verify that I understand what the code needs to do before diving in and coding the process!